HW-0 Solution

Scala 2018

Fibonacci numbers

```
fib 1 = 0
fib 2 = 1
fib n = fib (n - 1) + fib (n - 2)
```

So, solution is:

```
def fib: Int => Int = {
    case 1 => 0
    case 2 => 1
    case n => fib(n - 1) + fib(n - 2)
}
```

Problem of the easiest solution

```
fib(6) ==

fib(5) + fib(4) ==

fib(4) + fib(3) + fib(3) + fib(2) ==

fib(3) + fib(2) + fib(2) + fib(1) + fib(2) + fib(1) + 1 ==

fib(2) + fib(1) + 1 + 1 + 0 + 1 + 0 + 1 ==

1 + 0 + 4 ==

5
```

For big fib could be expanded to much

Improved solution

```scala
import scala.annotation.tailrec


def fibTailRec(n: Int): Int = {
  @tailrec def go(counter: Int, prev: Int = 0, next: Int = 1): Int = counter match
{
    case 1 => prev
    case _ => go(counter - 1, next, next + prev)
  }

  go(n)
}
```

Calculated like this

```
fib(6) ==

go(6, 0, 1) ==

go(5, 1, 1) ==

go(4, 1, 2) ==

go(3, 2, 3) ==

go(2, 3, 5) ==

go(1, 5, 8) ==

5
```

Calls like this optimized by a compiler

Tail call optimization or last call optimization

Prime numbers

```scala
 def prime(j: Int): Int = {
    def filterPrimes(ns: Stream[Int]): Stream[Int] = ns match {
      case n #:: rest => n :: filterPrimes( for (x <- rest if x % n != 0) yield x)
    }
//     filterPrimes(Stream.from(2))(j) - (j) takes js element from stream
//     cmp(filterPrimes)(Stream.from)(2)(j)
    (filterPrimes _ compose Stream.from)( 2)(j)
  }
def cmp[A, B, C](f: B => C)(g: A => B): A => C = f compose g
```

Implicit conversion

Streams does not supports :: operator as List, we can add this behaviour as:

```
class RichStream[A](str: =>Stream[A]) {
  def ::(hd: A) = Stream.cons(hd, str)
}
implicit def streamToRichStream[A](str: => Stream[A]): RichStream[A] =
  new RichStream(str)
```

Sum function

```
def sum(f: Int => Int, a: Int, b: Int): Int = {
    if (a > b) {
      0
    } else {
      f(a) + sum(f, a + 1, b)
    }
  }

def id: Int => Int = identity
def identity[A](x: A): A = x
```

Map function

```scala
def map[A, B](f: A => B, list: List[A]): List[B] = list match {
    case head :: tail => f(head) :: map(f, tail)
    case Nil => Nil
  }
```

Helpful for data-transformation instead of cycles

Scala version a bit harder, but more efficient, could be found at List class and TraversableLike trait

Filter function

```
def filter[A](p: A => Boolean, list: List[A]): List[A] = list match {
    case head :: tail => if (p(head)) {
      head :: filter(p, tail)
    } else {
      filter(p, tail)
    }
    case Nil => Nil
  }
```

We can sequence it with map to achieve more interesting List

FoldLeft and FoldRight just in case:)

```
def foldLeft[A, B](acc: B, f: (B, A) => B, list: List[A]): B = list match {
    case head :: tail => foldLeft(f(acc, head), f, tail)
    case Nil => acc
  }
```

foldLeft is useful for left-associative 'f', consider (-), also it tail-recursive

```
def foldRight[A, B](acc: B, f: (A, B) => B, list: List[A]): B = list match {
    case head :: tail => f(head, foldRight(acc, f, tail))
    case Nil => acc
  }
```

foldRight is useful for right-associative 'f', consider (::), also cool for lazy-lists

Execution of folds

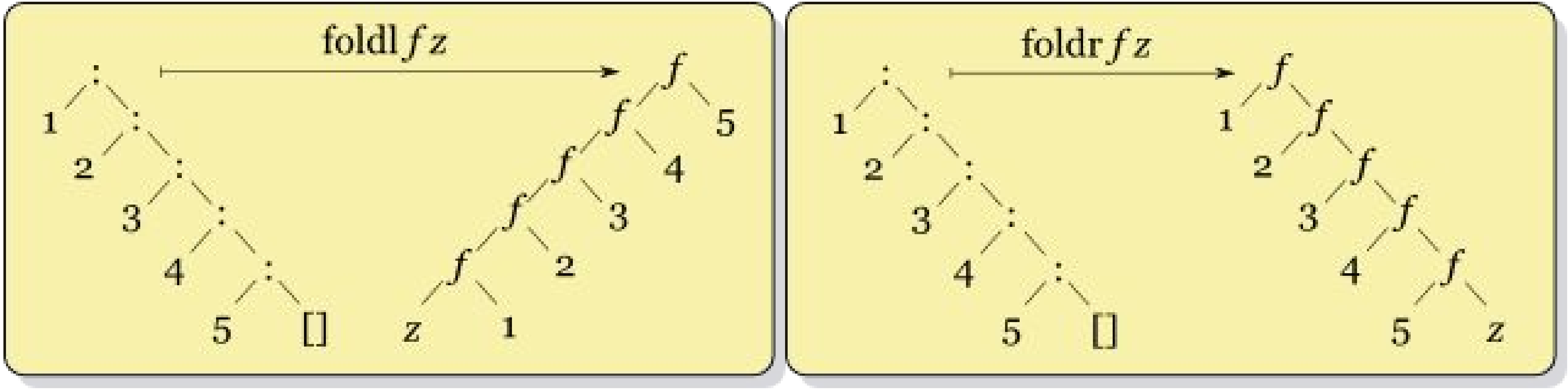
```
val ns = List(1, 2, 3, 4)
val f = (a: Int, b: Int) => a - b
ns == 1 :: 2 :: 3 :: 4
```

```
foldLeft(0, f, ns) ==
f(f(f(f(0, 1), 2), 3), 4) ==
((((0 - 1) - 2) - 3) - 4) ==
-10
```

```
foldRight(0, f, ns) ==
f(1, f(2, f(3, f(4, 0)))) ==
(1 - (2 - (3 - (4 - 0)))) ==
-2
```

Execution of folds - 2

https://en.wikipedia.org/wiki/Fold_(higher-order_function)



Map and filter using fold

```scala
def foldMap[A, B](f: A => B, list: List[A]): List[B] =
  foldLeft(Nil, (acc: List[B], el: A) => f(el) :: acc, list).reverse

def foldFilter[A](p: A => Boolean, list: List[A]): List[A] =
  foldLeft(Nil, (acc: List[A], el: A) => if (p(el)) {
    el :: acc
  } else {
    acc
  }, list).reverse
```

Look at ‘::’ and ‘.reverse’ at the end, if we use foldRight, then no need for reverse at the end

Natural numbers

```scala
sealed abstract class Nat

case object Zero extends Nat
case class Succ(n: Nat) extends Nat
```

```scala
def add(x: Nat, y: Nat): Nat = (x, y) match
{
  case (Zero, n) => n
  case (Succ(n), n1) => add(n, Succ(n1))
}

def sub(x: Nat, y: Nat): Nat = (x, y) match
{
  case (n, Zero) => n
  case (Succ(n), Succ(n1)) => sub(n, n1)
}
```

Last task

```scala
object FixCompile extends App {
  val mapper = (i: Int) => if (i % 2 != 0) i * 2 else i

  val result = List(1, 2, 3, 4, 5, 6, 7, 8, 9).map {
    mapper
   }.foldLeft(0) { (acc, v) => acc + v }

  print(result)
}
```